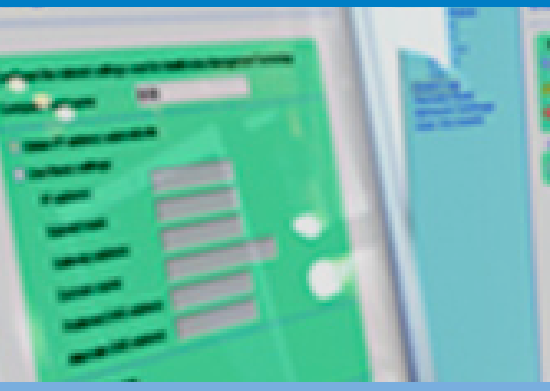
Volume 11 Issue 04 Published, November 15, 2007 ISSN 1535-864X DOI: 10.1535/itj.1104.04



Intel® Technology Journal

Multi-Core Software

Intel® Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation

Intel® Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation

Ilya Burylov, Performance Library Lab, Intel Corporation
Michael Chuvelev, Performance Library Lab, Intel Corporation
Bruce Greer, Performance Library Lab, Intel Corporation
Greg Henry, Performance Library Lab, Intel Corporation
Sergey Kuznetsov, Performance Library Lab, Intel Corporation
Boris Sabanin, Performance Library Lab, Intel Corporation

Index words: mathematics, library, parallel software, multi-core, vector math, BLAS, LAPACK

ABSTRACT

In this paper we present the Intel® Math Kernel Library (MKL) as a mathematical software package for scientific and technical computation designed for ease of use in environments that can vary greatly. Ease of use includes the build environment (use with different compilers), optimal performance on multiple platforms (automated selection of code based on the end-user system), optimal performance (optimization of an algorithm), interfaces to other libraries (FFTW), and effective use of multi-core processors through parallelization. We also discuss how this concept of ease of use will be expanded to provide more flexibility in the use of the library without greatly expanding its size.

Much of the paper is devoted to the optimization and parallelization of the library, critical in this era of multi-core processors. We discuss some of the methods used to improve performance that largely focus on cache utilization and minimization of table look-aside buffer (TLB) misses. Specifically, we look at the parallel performance of Basic Linear Algebra Subroutines [3] (BLAS), LAPACK [1], the Vector Math Library (VML), and a sparse linear solver (PARDISO). We include a brief section on a second application library, Integrated Performance Primitives (IPP), which complements the MKL in media applications.

INTRODUCTION

The Intel® Math Kernel Library (MKL) is a math library for use in scientific and engineering applications supporting a number of different mathematical areas:

Linear algebra. Basic Linear Algebra Subroutines (BLAS), LAPACK, ScaLAPACK, sparse BLAS, iterative sparse solvers, preconditioners, direct sparse solver (PARDISO)

Signal processing. FFTs, cluster FFTs

Vector math. Vector Math Library

Statistics. Vector Statistics Library with random number generators

PDEs. Poisson, Helmholtz solvers, trigonometric transforms

Optimization. Trust region solvers

Other. Interval linear solvers, multi-precision integer arithmetic

Among the key guidelines for the development of the library are using optimized math software for computationally demanding algorithms; threading and parallelizing these algorithms to make full use of multi-processor, multi-core [2], and multi-computer systems, making the library easy to use, and maintaining a high quality. Our focus in this paper is mostly on performance but we also introduce the paper with a discussion on ease of use.

A number of the features of the library do not relate to math functionality but contribute to ease of use. Some of these are:

- Designing the library to be compiler-independent eliminates the need for compiler-specific versions and allows C language programs to link to the Fortran portions of the library without the usual Fortran run-time libraries. Perhaps it is more correct to state that all compiler dependencies have been isolated (as will be explained in the discussion of the layer model of the library).

- Providing competitive performance on non-Intel® processors so software vendors can use a single library in their products for Intel® architecture computers.
- Parallelizing those parts of the library where parallelization makes sense. Most of the library functions could be parallelized but would not improve in performance if parallelized. Most of this paper deals with parallel performance on multi-core processors.
- Using interface files to map FFTW to MKL FFTs, other files to map older MKL FFTs to the more recent FFTs as well as using Java interface examples for various parts of the library.

To further enhance usability, future versions of MKL will introduce a "layer model" (see Figure 1). This version will have four layers: interface, threading, computational, and run-time, or compiler-specific, library layer.

The first layer already exists for the 32-bit Windows* version but will be ubiquitous in the library. This layer allows MKL to accommodate different interfaces, including, for instance, gfortran. This and some other Fortran compilers handle complex return values differently than the Intel compiler for the Intel® 64 Architecture-based processors on Linux*. This difference can be dealt with through an interface file without duplicating the rest of the library. Similarly, the basic library for a 64-bit operating system (OS) will use 64-bit integers going forward, but LP64 (32-bit integers for a 64-bit OS) will be accommodated with a layer.

An area that has been problematic, and will be more difficult going forward, has been the intermingling of user threaded code with MKL, where the user's program is compiled with a non-Intel compiler. The second layer deals with this mismatch. All MKL threading is function based, so the threaded portion will be compiled with different compilers (Intel and gfortran, for instance) and the threaded portion provided as a layer. By turning threading off during compilation of the threaded software, a non-threaded layer will create a sequential version of the library. By linking in the appropriate threaded layer, multiple threading environments will be supported, including a sequential version of the library, with just a small increase in the size of the package.

The third layer is the computational layer. This layer does all the computations and includes processor-specific code that is chosen at run time.

The fourth layer contains support files such as libguide, the threading library for Intel® compilers, and the BLACS, which are specific to compilers and message passing interface (MPI) versions.

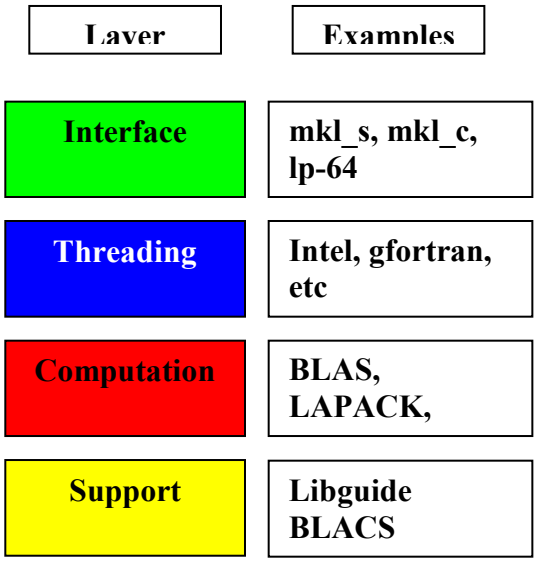


Figure 1: Layer model for MKL

In the rest of this paper we focus on performance for multi-core processors. Fortunately, many of the methods needed to achieve scaling with multi-core processor systems are similar to those used in shared memory parallel systems, at least for many of the functions of MKL. However, because of the shared caches of multi-core processors there are additional opportunities for threading functions such as VML, as explained in one of the performance sections.

We discuss parallelization and optimization for several different areas supported by the Intel® libraries in this order: BLAS, LAPACK, sparse linear solvers, VML, and codecs from IPP. Other key functions such as FFTs are not discussed. Especially in the cases of the BLAS and LAPACK, the contribution of the MKL developers is to take extant code and optimize it, including parallelizing it where that makes sense.

The fundamental problem for much mathematical software is how to structure the problem in such a way that the caches can be effectively used. Before looking at these problems it is useful to look at the problem from a data consumption versus data supply rate point of view.

Consider the Intel® Core™2 Duo processor, with a dual core running at 3.0 GHz performing the dot product. If we assume that one vector can be kept in cache, at what rate must the memory system supply data to keep just one dual-core processor busy? Each processor can do two double-precision multiplies per clock or four multiplies per clock, requiring 32 bytes (8 bytes per double precision word) per clock. At 3 GHz, this is 96 GB/second. For a dual-socket system (Woodcrest) the system must provide 192 GB/s to keep all four cores busy. On a Clovertown system the number of cores doubles again and the demand, at the same frequency, goes to 384 GB/s.

Choose any realistic memory bandwidth and divide it into the rate at which the processor can consume the data and you will have an estimate of the number of times a datum must be used once it is in cache in order to keep all the cores busy.

Much of the optimization efforts of MKL are centered on how to get that reuse factor high as well as how to deal with the many architectural complexity issues. In the following sections we discuss some of the problems and solutions for performance in MKL and briefly in IPP.

BLAS

Libraries are often an easy way to improve performance of an application. In the Introduction, we discussed the broad range of functionality that MKL offers as well as some of the ways the design is intended to make its use easy. Applications linked with MKL will see improvements in performance, especially if run on multi-core systems, through the many threaded functions of the library.

The real issue is how MKL takes advantage of performance features such as SIMD hardware, and why multi-core processing exacerbates performance-sensitive issues. We start by describing single core performance optimization and move onto parallelization. If the single-core performance is far from optimal, it logically follows that the multi-core performance may not be ideal either.

Were MKL limited to a single set of functions, that set would be the BLAS because of its importance as a building block for higher order linear algebra functionality. The BLAS encapsulates several important dense linear algebra kernels.

"Levels" is an important notion of the BLAS philosophy. Examples of Level 1 algorithms include taking an inner product of two vectors, or scaling a vector by a constant multiplier. Level 2 algorithms are matrix-vector multiplication or a single right-hand-side triangular solve. Level 3 algorithms include dense matrix-matrix multiplication. If we assume a vector is length N or a matrix is order N, then the number of floating point operations (flops) for a Level 1, Level 2, and Level 3 algorithm are O(N), $O(N^2)$, and $O(N^3)$, respectively. The data movement, however, is O(N), $O(N^2)$, and $O(N^2)$, respectively. This last fact is crucial for optimization and threading performance. This makes the number of floating point operations per data item moved O(1), O(1), and O(N), respectively.

Memory performance is inadequate to directly support the computational speed of the processor. This gap has increased over the years and multi-core processors accelerate the mismatch between memory system performance and the data demands of the processor. To deal with this discrepancy, processors use a memory hierarchy. Each level of the memory hierarchy boasts a different latency and bandwidth. We consider the highest level machine registers. Register data movement keeps pace with processor clock rates. The next level is the first-level (L1) cache (small size) followed by the second-level (L2) cache (larger). Some machines also have a third-level (L3) cache (largest). Finally, at the bottom, there is the machine memory. This is often pictured as a pyramid, as shown in Figure 2.

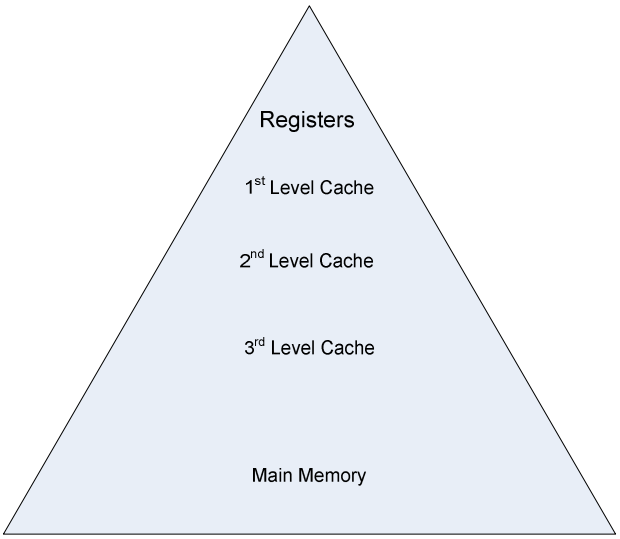


Figure 2: Memory hierarchy pyramid

The closer to the top of the pyramid, the more valuable the resource is and the greater its performance in terms of bandwidth and reduced latency. The challenge for the developer is to keep data in the fast memories long enough to amortize the cost of getting the data there. Blocking algorithms along with data organization ensure that more work gets done at the faster top of the pyramid. MKL blocks algorithms such as the Level 3 BLAS, where the amount of work, $O(N^3)$, can be much greater than the amount of data movement, $O(N^2)$.

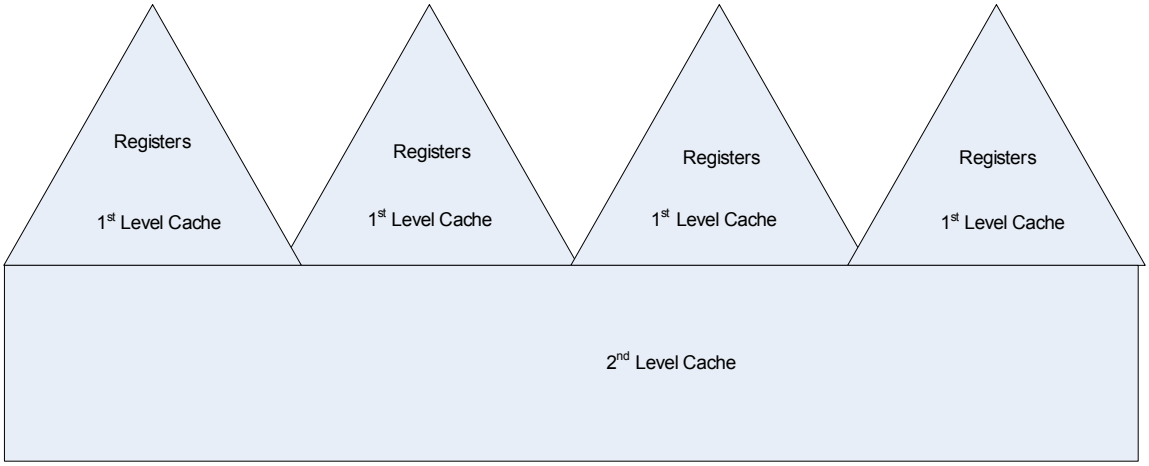


Figure 3: Shared cache top of pyramid

While this situation has existed in architectures for many years, the recent advent of multi-core processing merely adds to the complexity of the problem both because parallelism is not mandatory and because of the sharing of caches between cores. If two or more cores share a secondary cache, for instance, the familiar top of the pyramid suddenly looks like Figure 3. The bottom of the pyramid remains the same as in Figure 2.

What we typically see is a large cost for moving elements from main memory, compared to the very fast capacities of Figure 2. The complexity of the memory system—mapping the memory onto the cache—includes the use of an additional cache called the table look-aside buffer, or TLB. Each memory page mapped to the cache has a TLB entry.

When data are referenced, they may be in the L1 cache, L2 cache, or in memory. In addition, the page may or may not be in the TLB. Each miss—L1, L2, higher order cache, TLB—is increasingly expensive to retrieve. While the processor can hide some cache misses, TLB misses will cause stalls while the page address for the data is found and loaded.

In addition to the cache structure we have already outlined, most caches have a given associativity set, meaning how addresses are shared in their mapping to a given cache line. There is also a dependency on the cache replacement policies that determines when cache lines are evicted from the cache. There are other features of caches that will affect the performance of the processor: bank structure, how and when data are written back to memory, and so on.

All of these issues are accounted for either explicitly (by design) or implicitly (by automated searches through design space) for key MKL functions such as the BLAS. The result is code that is tuned for a single core. Now we need to parallelize the code.

One of the most important considerations is where to thread an application. If an algorithm from LAPACK calls the Level 3 BLAS, there is now a choice of where to thread. One can parallelize at the LAPACK level, the BLAS level, or both. We have consistently found it to be the case where parallelizing at the LAPACK level yields the greatest advantage.

Figure 4 illustrates this for the LAPACK function DGETRF, which performs LU factorization and is the basis for the LINPACK benchmark. The chart shows the ratios of performance for threading at the LAPACK and BLAS levels, with the BLAS-level performance being 1.0. Problem sizes are 1,000 times the abscissa values.

As the figure shows, for smaller sizes, the higher-level threading is up to 80% faster. But even at 30,000 equations, the LAPACK-level threading is nearly 10% faster on eight threads and 5% faster on two and four threads.

LAPACK

In the previous section we discussed the factors that go into the optimization of functions and showed how choosing the right level for parallelization can have a substantial impact on parallel performance as the number of cores increases, using LU factorization (DGETRF) as an example. The MKL has threaded and optimized many of the most important LAPACK functions. The problem is usually the same: how to feed the arithmetic units, which translates into how to get data into the caches and then to reuse them sufficiently to accommodate the substantial

differences in the rate of consumption by the floating point hardware and the rate of supply by the memory subsystem.

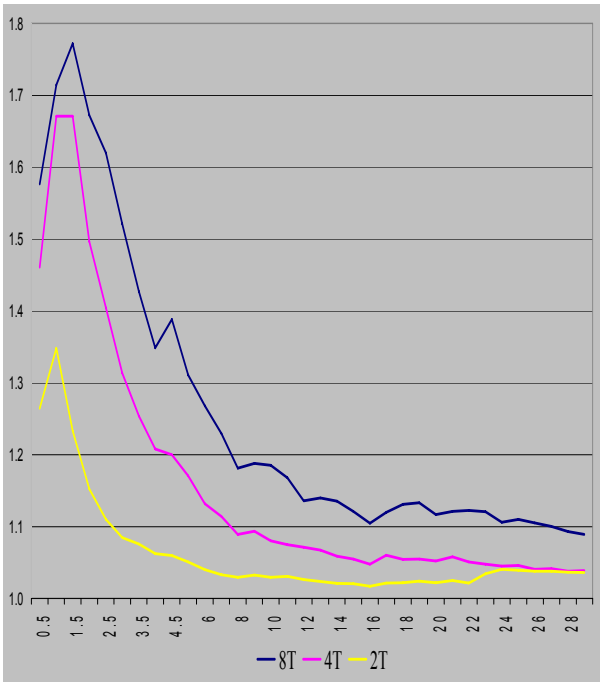


Figure 4: LAPACK vs. BLAS-level threading

LAPACK largely replaced LINPACK and employs blocked algorithms instead of the vector algorithms of LINPACK, making it much better suited for cache-based architectures. However, there are many areas where LAPACK code can further employ Level 3 BLAS [4] instead of the lower-level functions, which can improve cache usage. That, in turn, improves parallel performance, including performance on multi-core systems. We provide several examples of how increasing the use of higher-level BLAS substantially improves the performance of the MKL implementation of LAPACK over the reference implementation.

One of the important linear algebra applications is double-sided decompositions like singular value decomposition (SVD) or Symmetric Eigenvalue Decomposition. In MKL we block the chains of plane rotations using Level 3 BLAS, resulting in remarkable improvements in performance of up to about 18x. Figure 5 compares the resulting threaded symmetric solver DSYEV against the reference implementation, with performance improvements of up to approximately 18x. In this chart, the MKL performance[1] is threaded using eight threads, computing all eigenvectors.

A second example employing a blocking algorithm implementation that allows the use of higher-order BLAS are the routines operating on packed storage format. This optimization requires the allocation of additional workspace of size N*NB (where N is the size of the problem, and NB is the block size, usually around 64). Use of workspace is common in other LAPACK functions and the cost, in terms of memory usage, is small.

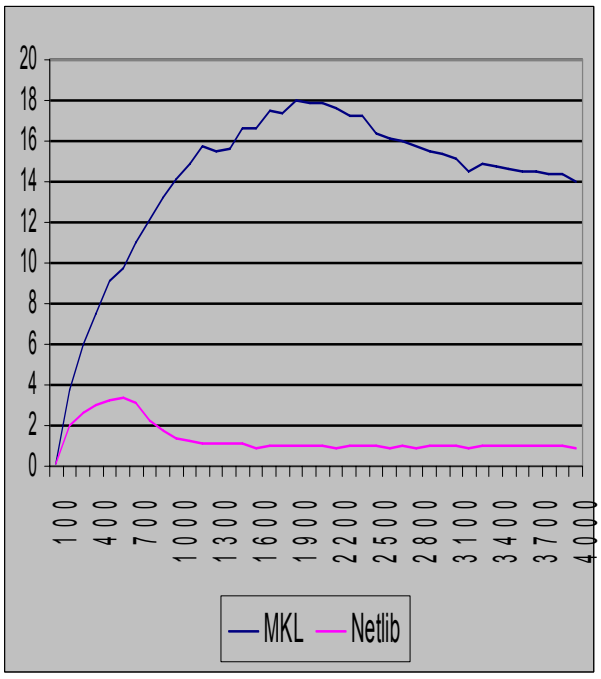


Figure 5: DSYEV improvements via Level 3 BLAS

In the case of the Cholesky solver performance on packed storage format, the performance improvement again is around 18x on the same system as for DSYEV, as shown in Figure 6.

While restructuring of the LAPACK code to use Level 3 BLAS improves performance markedly, more advanced techniques must be employed to minimize dependencies on the sequential code that remain after employing Level 3 BLAS.

[1] 2.4 GHz, Dual-socket, Quad-Core Intel® Xeon® processor 5300 Series 1067 MHz front-side bus. 2x4 MB L2 cache.

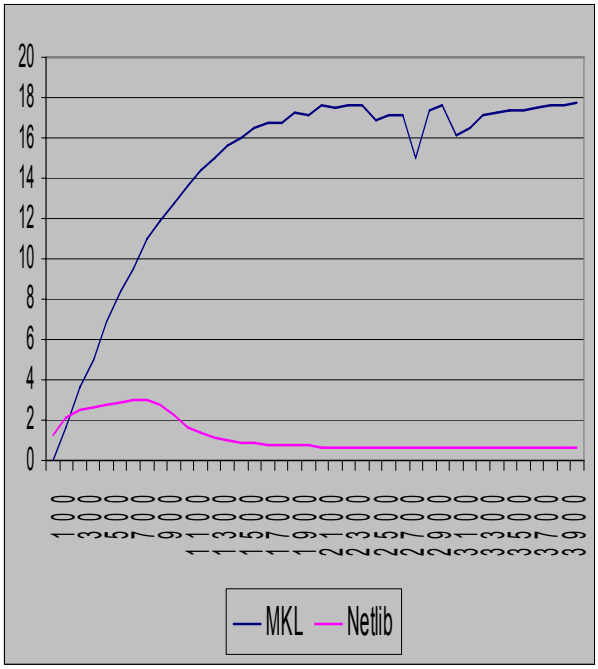


Figure 6: Packed-format Cholesky factorization

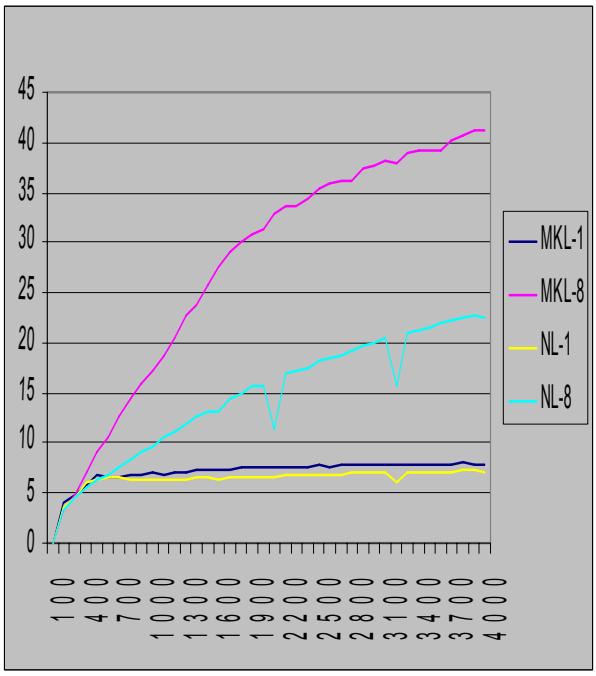


Figure 7: DGETRF-level versus BLAS-level threading

In such functions as LU and QR factorization [5], a look-ahead technique is used that allows the next block factorization to begin before the matrix has been fully updated, which increases concurrency. Figure 7 looks at DGETRF performance on an 8-core system comparing MKL versus netlib performance. As the chart shows, there are optimizations in MKL that improve the performance even on one thread vis-à-vis the reference implementation.

VECTOR MATH LIBRARY (VML)

As we suggested earlier, the main issue in threading various math functions is not so much whether they can be threaded (are operations separable) but rather whether there are sufficient operations on the data once they are in cache to permit other cores/processors to also get data to work on. In other words, this all comes down to a memory bandwidth issue.

The transcendental functions of VML typically require 10-50 cycles per element (CPE), typically with one input and one output value per element. Taking this into consideration we can roughly estimate the break-even point of threading by the following inequality:

$$S/T+O < S \rightarrow N*CPE/T+O < N*CPE \rightarrow N > O * T / (CPE * (T-1)),$$

where S = CPE*N and N is the vector length – is the number of cycles to execute a particular function in serial mode, O is the number of clocks for overhead for starting threads (it really depends on T, the total number of threads used) and CPE which is the cpe of the function in serial case. One can see that with increasing CPE (more complex functions) the shorter vectors can be effectively parallelized. The greatest difficulty here is to make an estimation of O.

Our computations show that O, measured in cycles, depends mostly on the number of sockets, the number of cores, and whether hyperthreading is turned on or off. This inequality, estimation of O, and a table of CPE values for each function are used in order to choose the number of threads for a particular function call during runtime.

Figure 8 shows the speedups for three VML functions on a Woodcrest system (dual socket, dual core) compared to single-thread performance on the same processor.

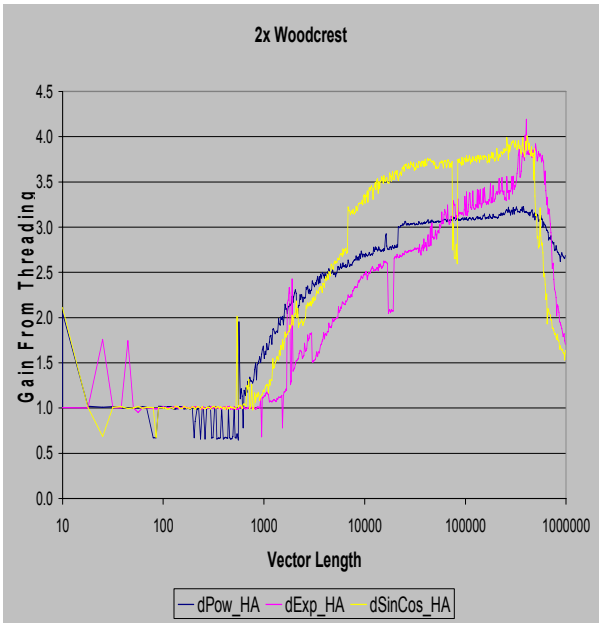


Figure 8: VML scaling on selected functions

Though VML can choose some particular number of threads it is difficult to do this accurately:

- *Performance is often data dependent*. For example, the dCbrt function (cubic root for double precision vectors) has 27.96 cpe for uniformly distributed data on the interval [-10000;10000], but 15.00 cpe if the vector is all zeros.
- *Data location*. If the input/output vectors are in cache, scaling and performance will be much better than if the data are in memory.
- When several different successive vector functions work with the same vectors, a different number of threads can be chosen, and as a result data might stay in the wrong cache if the cache is not shared.
- The influence of overhead might be significantly lowered by using threading at a higher level (i.e., if the user calls the VML functions from a threaded application).

In summary, for VML, multi-core shared cache architectures have opened opportunities for threading that did not exist previously, but the performance is dependent on factors the VML developer can only partly control. It is likely that in most cases calling VML functions from a threaded application will result in better performance than invoking the threaded VML.

SPARSE LINEAR ALGEBRA

Solving large sparse linear systems of equations is often a stumbling block in many scientific problems. MKL offers several approaches for solving such problems. The PARDISO solver is a sparse direct supernodal solver that is thread-safe, high-performing, and memory-efficient. Using this tool, you can solve symmetric and non-symmetric sparse linear systems of equations on shared-memory multi-processors. However, there is a point where the memory requirements for large systems can become prohibitively high, and the PARDISO/DSS (direct sparse solver) will not work. This is where MKL iterative sparse solvers come in: these solvers can provide a remedy, because only a few working vectors and the primary data need be stored.

MKL iterative solvers are based on a reverse communication interface (RCI) scheme that makes the user responsible for providing certain operations for the solver (for example, matrix-vector multiplications). To simplify the usage of MKL iterative solvers and gain additional performance, MKL offers sparse BLAS functions, which is a set of functions that perform a number of common vector and matrix operations for the most popular sparse storage schemes: compressed sparse row (CSR), compressed sparse column (CSC), diagonal, coordinate (COO), skyline, and block sparse row formats. Most MKL sparse BLAS routines are threaded using OpenMP. As in the case of the VML, for instance, performance on sparse BLAS is improved when the data are in the common cache for the cores and those BLAS are threaded.

Like dense matrices, the performance of MKL PARDISO/DSS and MKL sparse BLAS depends on the details of the machine architectures, but unlike dense problems, the performance of these components also depends on the structure of the matrix, because the distribution of the nonzero elements in a sparse matrix determines the memory access patterns. However, many physical problems expose a well-behaved sparse structure, or the rows can be re-ordered to yield a better structure. PARDISO uses approximate minimum degree ordering and METIS reordering techniques for getting permutations to minimize fill-in and the associated memory requirements. Internal storage for the matrix factors in PARDISO is a block format. Most of the computations are done with the help of MKL Level 3 BLAS and LAPACK. The usage of Level 3 BLAS and supernode pivoting coupled with supernode partitioning and synchronous computations allows PARDISO to achieve high-gigaflop rates and nearly linear speedup on multi-core platforms.

There are differences in optimization of Level 2 and Level 3 sparse BLAS on many core platforms, and some

optimization problems are similar to the problems of dense Level 2 and Level 3 BLAS (e.g., low locality in Level 2 routines). For Level 3 sparse BLAS, reorganizing the computations to perform the entire set of multiplications as a single operation produces significantly better performance. It is natural to expect that performance and scalability of Level 3 sparse BLAS are better than those of Level 2 sparse BLAS. MKL sparse BLAS routines for the block sparse row format that exploit the benefits of data blocking have better data locality and vector instructions: for example, the SSE2 instruction set can be applied even for Level 2 sparse BLAS in this case. Similar optimizations are done for the diagonal and skyline format, because the elements of the source vector as well as destination vector are accessed sequentially. The Level 2 sparse BLAS operations for point entry sparse formats, such as the compressed sparse row (CSR) or coordinate formats (COO), are the most difficult area for optimization, because the elements of the source vector are accessed in a discontinuous way that leads to poor temporal locality. However, it appeared that even Level 2 sparse BLAS can be threaded effectively at least on the latest Intel® multi-core platforms. In addition, many well-known methods have been used for optimization of MKL Sparse BLAS. Among these are blocking, prefetching, OpenMP, etc., which allow for better performance of Sparse BLAS on multi-core architectures.

INTEGRATED PERFORMANCE PRIMITIVES (IPP)

IPP is a multi-functional library highly optimized for Intel architecture. IPP covers 15 functional domains that can be recognized by a suffix in the library file names. For example, functions with IPPs in their names are signal processing functions, note suffix "s." There are more than two-thousand functions processing 1D signals/data of different data types: real and complex, signed and unsigned, floating point, and integer. The other libraries in IPP are image processing "i," JPEG primitives "j," audio coding "ac," color conversion "cc," string processing "ch," cryptography "cp," computer vision "cv," data compression "dc," small matrix operations "m," realistic rendering "r," speech coding "sc," speech recognition primitives "sr," video coding "vc," vector math "vm."

IPP is optimized for several Intel architectures: IA32, IA64, Intel® 64, and IXP. Within each architecture are optimizations for specific processors. For instance, within IA32 architecture there are specific optimizations for the Pentium® 4 and Intel Core 2 Duo processors, among others.

IPP is optimized at three levels: algorithmic, effective use of SIMD instructions (SSE2, SSE3), and parallelization at both the primitive and component levels. Primitive-level threading is the threading implemented in IPP functions. Not every function in IPP is parallelized because of the overhead added by threading. However, the good news here is that IPP is by design a set of build blocks and applications that developers can easily use to thread their application by calling the primitives on different threads.

Component-level threading is threading provided in such components as video codecs, the H264 encoder and decoder; the jpeg viewer, and the IPP implementation of well-known data compression libraries, ZLIB and GZIP. These components, as well as others, are shipped with IPP as IPP samples given in their source codes.

An example of algorithm optimization is the median filter in the Signal and Image processing domains. Table 1, for instance, illustrates the results, in clocks-per-element, of IPP optimization of the median filter compared with the LEADtools library.

Table 1: IPP compared to LEADtools on median filter

Spatial filter with mask 5x5	Function	cpe
LEADtools	L_MedianFilterBitmap	345
IPP	ippiFilterMedian_16s_C3R	35

CPU optimization with the SIMD instruction set, which is done for many functions in IPP, also gives a performance gain that can be measured by comparing the performance ratio numbers of the C version of the library to the CPU specific library, such as optimizing for the Intel Core 2 Duo processor. Table 2 illustrates the performance advantage of multi-core threading on MPEG4 decoding.

Table 2: Speedup on threaded MPEG4

Stream[2]	Resolution	Frames	Bitrate MB/s	FPS 1T	FPS 2T	Ratio
1	1280x720	IPB	4.0	199	328	1.65
2	720x576	IP	4.7	298	411	1.38
3	640x476	IP	2.2	671	841	1.25
4	640x480	IP, OBMC	1.0	650	650	1.6

[2] Stream 1: preakness_59.94fps_Xvid_4Mbs_CBR.avi; Stream 2: Boss.avi; Stream 3: Taxi.avi; Stream 4: Term2.divx

SUMMARY

The Intel MKL is part of a suite of tools offered by Intel to help developers create software efficiently and to achieve high performance. For MKL, the goal has been to provide an easy-to-use software package to aid in the development of mathematical software. Achieving that goal has a number of facets, some of which we have touched on in this paper: functionality, compiler independence, performance, and the most recent efforts in performance, focusing on helping the user get the full benefits available from Intel multi-core systems. We have discussed in general terms some of the approaches taken by library developers to achieve the performance goals including threading at a higher level of functionality (LAPACK) and improving the locality of reference for data in LAPACK codes through more effective use of the Level 3 BLAS, and so on.

As the complexity and core counts for microprocessors continue to grow, MKL (and IPP) will optimize functions that impact performance in key application areas ensuring full and effective use of those processor developments.

ACKNOWLEDGEMENTS

Neither MKL nor IPP would be possible without the creative and disciplined efforts of the teams that develop these software packages. We acknowledge their contributions that make these features, functions, and performance possible. Both libraries have been largely created by groups of developers with strong backgrounds in computer science and mathematics in Russia.

REFERENCES

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenny, A., and Sorenson, D., *LAPACK User's Guide. 3rd ed*., SIAM, Philadelphia, PA., 1999.

[2] Buttari1, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P. and Tomov, S., "The Impact of Multicore on Math Software," at http://icl.cs.utk.edu/projectsfiles/sans/pubs/para-multicore-2006.pdf

[3] Dongarra, J., DuCroz, J., Duff, I. Hammarling, S., "A set of Level 3 Basic Linear Algebra Subprograms," *Technical Report, ANL-MCS-TM-88*, Argonne National Laboratory, Argonne, ILL, 1988.

[4] Lang, B, "Using Level 3 BLAS in Rotation-Based Algorithms," *SIAM Journal on Scientific Computing*, Volume 19, Number 2, pp. 626–634, 1998.

[5] Starzdins, P., "A comparison of lookahead and algorithmic blocking techniques for matrix factorization," *TR-CS-98-07, The Australian National University*, July 1998.

AUTHORS' BIOGRAPHIES

Ilya Burylov is a Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. His interests include multi-core threading technologies, numerical analysis, and partial differential equations in hydrodynamics. He received his M.S. degree from the Perm State Technical University. His e-mail is ilya.burylov at intel.com.

Michael Chuvelev is a Senior Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. He graduated from Moscow Institute of Physics and Technology with an M.S. degree in Applied Mathematics. At Intel he has specialized on code parallelization and linear algebra software optimization, particularly of LAPACK, ScaLAPACK, and sparse solvers. He is currently focusing on LAPACK optimization on SMP systems, especially on multi-core systems. His e-mail is michael.chuvelev at intel.com.

Bruce Greer is a Principal Engineer in the MKL team in the Developer Products Division of the Software Solutions Group. He was manager of MKL from 1995 until May of this year. He received an M.S. degree in Physics from Georgia Tech. His professional interests are in code optimization. Despite his handicap, he has a passion for golf. His e-mail address is bruce.s.greer at intel.com.

Greg Henry is a Principal Engineer in the MKL team in the Developer Products Division of the Software Solutions Group. His research interests are linear algebra, parallel computing, numerical analysis, scientific computing, and all things relevant to MKL. He received his Ph.D. degree from Cornell University in Applied Mathematics and started working at Intel in August 1993. Greg has three children and a wonderful wife, and writes novels as a hobby. His e-mail is greg.henry at intel.com.

Sergey Kuznetsov is a Senior Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. His research interests include parallel numerical linear algebra, sparse matrix computations, parallel algorithms and eigenvalue problems. He received his Ph.D. degree from the Novosibirsk State University, Russia. His e-mail is sergey.v.kuznetsov at intel.com.

Boris Sabanin is a Principal Engineer and the Engineering manager of the Intel Integrated Performance Primitives library in the Intel Systems and Solutions Group. He is focusing on the design, development and optimization of signal processing functions. His e-mail is boris.sabanin at intel.com.

GLOSSARY

BLACS: Basic Linear Algebra Communication Subprograms – a set of functions developed for ScaLAPACK which isolate the communications used by the software from the communication layer such as MPI. Used throughout MKL cluster software.

BLAS: *Basic Linear Algebra Subprograms* – a set of dense vector, vector matrix and matrix math functions useful in creating higher level functions such as solvers.

CODEC: COder/DECoder – used for encoding or decoding digital data streams such as video or audio.

DSS: Direct Sparse Solver – solves a system of equations in an *a priori* known number of operations in contrast to iterative sparse solvers for which the number of operations is data dependent.

FFT: Fast Fourier Transform – algorithms to convert, for instance, a time series into a frequency series in an efficient way.

FFTW: Fastest FFT in the West – a publicly available software package to create highly optimal FFTS. See http://www.fftw.org.

IMSL – A large commercial math software package. See http://www.vni.com

LAPACK: Linear Algebra PACKage – a set of solvers for systems of equations, eigensolvers, etc, using blocked algorithms that make effective use of the Level 3 BLAS.

LINPACK – Predates LAPACK and based on vector operations. Also a benchmark solving systems of linear equations.

METIS – A set of programs for partitioning unstructured graphs. See http://glaros.dtc.umn.edu/gkhome/views/metis

MPI: Message Passing Interface – A widely used distributed memory (cluster) communication package.

NAGLIB – A large math software package similar to IMSL. See http://www.nag.com

NETLIB – A repository of software packages such as BLAS, BLACS, LAPACK, LINPACK, ScaLAPACK and others. See http://www.netlib.org

PARDISO: PARallel DIrect SOlver – A parallel direct solver from University of Basel and licensed by MKL. See http://www.pardiso-project.org

PDE – Partial Differential Equation.

ScaLAPACK: Scalable LAPACK – cluster versions of much of LAPACK.

SIMD: Single Instruction Multiple Data – hardware to perform multiple arithmetic operations simultaneously on a single instruction, such as the SSE2 and SSE3 instructions.

For further information visit:

developer.intel.com/technology/itj/index.htm